Introduction to advanced parameter optimization

1. Introduction

Thus far, we have studied (1) basic feedforward neural networks, and (2) one basic training algorithm for adjusting the weights (parameters) in a neural network — namely, *gradient descent*, where the *backpropagation* algorithm is used to compute the gradient of the neural-network error with respect to the current weights $\mathbf{w}(t)$, $\nabla E[\mathbf{w}(t)]$, efficiently. Over the next few weeks, we will extend our study of neural networks by looking at (1) advanced training algorithms, and (2) adaptive neural-network architectures.

2. Heuristic extensions of gradient descent

A. Convergence in gradient descent

Recall that in the gradient-descent algorithm, we update weights at step t according to,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t) \tag{1}$$

where,

$$\Delta\mathbf{w}(t) = -\eta\nabla E[\mathbf{w}(t)], \tag{2}$$

$\mathbf{w}(t)$ are the weights of the neural network at step t, η is the fixed learning-rate parameter, and $\nabla E[\mathbf{w}(t)]$ is the gradient of the neural network error E with respect to the training data at $\mathbf{w}(t)$. There are two main problems with this algorithm: (1) slow convergence to a local minimum of the error surface, and (2) trial-and-error selection of the learning rate η.

In actual neural-network training, often 100,000s of steps, or *epochs*, of the gradient-descent algorithm are required to converge to a good local minimum. (An epoch is defined to be one complete presentation of the training data.) Advanced training algorithms that we will develop in the next several weeks can reduce this large number of required epochs by several orders of magnitude. Moreover, in some of these algorithms, we will not have to hand-pick critical parameters like the learning rate η; rather, all necessary learning parameters will be computed automatically as a function of the local second-order properties of the error surface.

Below, we will study the convergence properties of the gradient-descent algorithm through a specific example; namely, we will reconsider the simple, two-dimensional quadratic "error" surface that we looked at last time,

$$E = 20\omega_1^2 + \omega_2^2. \tag{3}$$

Note, of course, that this could not possibly be a real neural-network error surface, since a neural network typically has many more weights than just two; moreover, a neural-network error surface is almost never *globally* quadratic. Nevertheless, the lessons that we learn about the convergence properties of the gradient-descent algorithm for this simple error surface will inform our development of more advanced training algorithms. A *quadratic* error surface like that in (3) can be especially instructive, because *any* arbitrary error surface can be approximated as *locally* quadratic near a local minimum of that error surface. Just as a simple one-dimensional function $f(x)$ can be approximated by its second-order Taylor series about some point x_0,

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + f''(x_0)(x - x_0)^2 \tag{4}$$

an error function $E(\mathbf{w})$ can be approximated as,

$$E(\mathbf{w}) \approx E(\mathbf{w}_0) + (\mathbf{w} - \mathbf{w}_0)^T\mathbf{b} + \frac{1}{2}(\mathbf{w} - \mathbf{w}_0)^T H(\mathbf{w} - \mathbf{w}_0) \tag{5}$$

about some vector $\mathbf{w}_0$, where,

$$\mathbf{b} = \nabla E(\mathbf{w}_0), \text{ and,} \tag{6}$$

$$H = \nabla[\nabla E(\mathbf{w}_0)], \tag{7}$$

is the *Hessian* matrix of second partial derivatives evaluated at $\mathbf{w}_0$.

Definition: The $W \times W$ *Hessian* matrix H of a W-dimensional function $E(\mathbf{w})$ is defined as,

$$H = \nabla[\nabla E(\mathbf{w})] \tag{8}$$

where,

$$\mathbf{w} = \left[\omega_1 \ \omega_2 \ \ldots \ \omega_W\right]^T. \tag{9}$$

In other words, element (i, j) of the H matrix, is given by,

$$H_{(i,j)} = \frac{\partial^2 E}{\partial \omega_i \partial \omega_j} \tag{10}$$

In order to study the convergence properties of the gradient-descent algorithm, we will require a few definitions and results from linear algebra.

Definition: For a $W \times W$ square matrix H, the *eigenvalues* λ are the solution of,

$$\left|\lambda I_W - H\right| = 0 \tag{11}$$

where $|\bullet|$ denotes the determinant, and I_W denotes the $W \times W$ identity matrix.

Definition: A square matrix H is *positive-definite*, if and only if all its eigenvalues λ_i are greater than zero. If a matrix is positive-definite, then,

$$\mathbf{v}^T H \mathbf{v} > 0, \ \forall \mathbf{v} \neq 0. \tag{12}$$

For a quadratic error surface, H is constant and is positive-definite everywhere. For an arbitrary error surface, H is positive-definite near a local minimum of that error surface.

Now, let λ_{min} and λ_{max} denote the smallest and largest eigenvalues, respectively, of the Hessian H with respect to an error surface $E(\mathbf{w})$. Assuming that we are near a local minimum,

$$\lambda_{min} > 0, \tag{13}$$

the gradient descent algorithm converges maximally at a rate proportional to,

$$(\lambda_{min} / \lambda_{max}), \tag{14}$$

and the learning rate η is bounded by,

$$0 < \eta < \frac{2}{\lambda_{max}} \tag{15}$$

for convergent (as opposed to divergent) behavior. As an example, let us compute the eigenvalues of H for the error surface in (3). The first partial derivatives are given by,

$$\frac{\partial E}{\partial \omega_1} = 40\omega_1, \ \frac{\partial E}{\partial \omega_2} = 2\omega_2, \tag{16}$$

and the second partial derivatives are given by,

$$\frac{\partial^2 E}{\partial \omega_1^2} = 40, \ \frac{\partial^2 E}{\partial \omega_2^2} = 2, \ \frac{\partial^2 E}{\partial \omega_1 \partial \omega_2} = \frac{\partial^2 E}{\partial \omega_2 \partial \omega_1} = 0. \tag{17}$$

Therefore, the Hessian H is given by,

$$H = \begin{bmatrix} 40 & 0 \\ 0 & 2 \end{bmatrix} \tag{18}$$

For this matrix, the eigenvalues are easy to compute:

$$|\lambda I_2 - H| = 0 \tag{19}$$

$$\left|\lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 40 & 0 \\ 0 & 2 \end{bmatrix}\right| = 0 \tag{20}$$

$$\begin{vmatrix} \lambda - 40 & 0 \\ 0 & \lambda - 2 \end{vmatrix} = 0 \tag{21}$$

$$(\lambda - 40)(\lambda - 2) = 0 \tag{22}$$

$$\lambda_{min} = 2, \lambda_{max} = 40 \tag{23}$$

[Note: The determinant of a 2×2 matrix,

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \tag{24}$$

is given by $|A| = ad - cb$.]

From equation (15), the learning rate η is therefore bounded by,

$$0 < \eta < \frac{2}{40} = 0.05 \tag{25}$$

[Note that the bound in (25) is equivalent to the one we derived last time by directly looking at the gradient-descent weight iteration equations,

$$\omega_1(t+1) = \omega_1(t) - \eta \frac{\partial E}{\partial \omega_1(t)} \tag{26}$$

$$\omega_1(t+1) = \omega_1(t)(1 - 40\eta) \tag{27}$$

$$\omega_2(t+1) = \omega_2(t)(1 - 2\eta). \tag{28}$$

Both (27) and (28) are of the form,

$$\omega(t+1) = c\omega(t) \tag{29}$$

which converges to zero for any c such that,

$$-1 < c < 1. \tag{30}$$

Thus, from (27) we require that,

$$-1 < 1 - 40\eta < 1 \tag{31}$$

$$0 < \eta < 0.05 \tag{32}$$

since equation (28) leads to the weaker bound $0 < \eta < 1$.]

Figure 1 below plots the trajectory of the gradient-descent algorithm for three different learning rates $\{0.01, 0.04, 0.05\}$ and initial weights $(\omega_1, \omega_2) = (1, 2)$. Each trajectory is superimposed on top of a contour plot of E and is stopped when $\sqrt{E} < 10^{-6}$ (except for $\eta = 0.05$, where gradient descent fails to converge). From this simple example, we see that the learning rate η can be critical to the rate of convergence in gradient descent. An inappropriately small learning rate leads to slow convergence, while an inappropriately large learning rate can lead to divergence.

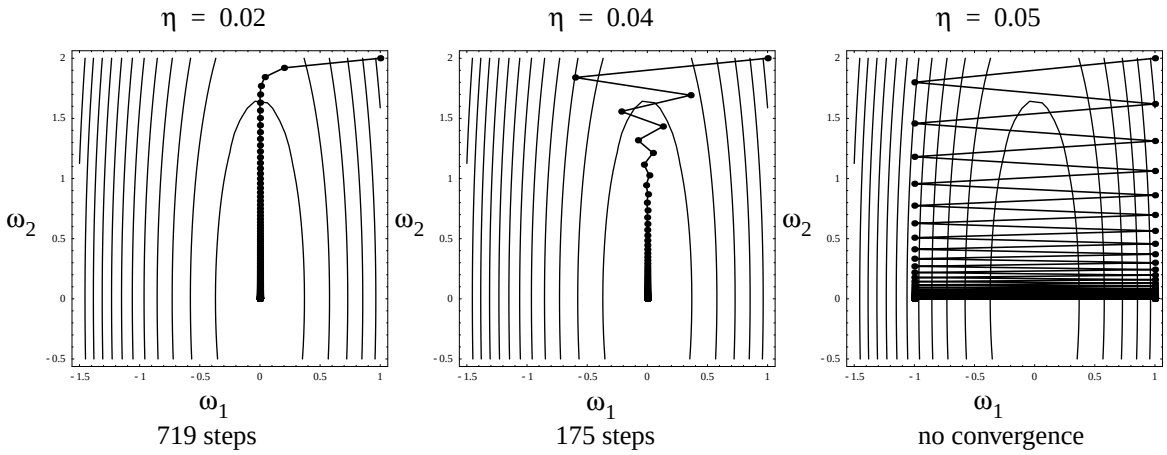


Figure 1

The gradient descent algorithm is especially vulnerable to slow convergence where the error surface consists of long valleys with steep sides, as in this example. We see that the weight ω_1 quickly converges to its appropriate value, while the weight ω_2 takes much longer to reach its optimal value of zero. Let us characterize mathematically what it means for an error surface to consist of a "long valley with steep sides." Geometrically, the lengths of the contour lines of the error surface in Figure 1 are proportional to,

$$1/\sqrt{\lambda_i}\,,\ i \in \{1, 2\}\,, \tag{33}$$

where λ_i are the two eigenvalues of H for the error surface in equation (3). In other words, "long valleys with steep sides" in the error surface correspond to small ratios in (14), and, consequently, slow rates of convergence. For regions of the error surface with small ratios,

$$(\lambda_{min}/\lambda_{max})\,, \tag{34}$$

rather than a single, fixed learning rate along all dimensions of the error surface, we instead need different learning rates for each weight. All the methods we will study from here on out can be viewed as varying attempts to implement different effective learning rates along different dimensions of the neural network error surface.

B. Gradient descent with momentum

One of the most often used heuristic modifications to basic gradient descent is the addition of a *momentum term*, which is intended to speed up convergence of gradient descent. In gradient descent with momentum, we update the weights at step t according to,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t) \tag{35}$$

where,

$$\Delta\mathbf{w}(0) = -\eta\nabla E[\mathbf{w}(0)] \tag{36}$$

$$\Delta\mathbf{w}(t) = -\eta\nabla E[\mathbf{w}(t)] + \mu\Delta\mathbf{w}(t-1)\,,\ t > 0\,,\ 0 \le \mu < 1 \tag{37}$$

Here, μ is known as the momentum parameter. Note that now, the change of the weights at step t is not only a function of the weights at t, but also of the weights at $(t-1)$. Ideally, the addition of the momentum term leads to higher effective learning rates in shallow dimensions of the weight space, while in steep dimensions of the weight space, it has relatively little effect. Consider first a shallow region of the weight space, where we can assume that the gradient is approximately constant such that,

$$\nabla E(\mathbf{w}_t) \approx \nabla E(\mathbf{w}_0) = \mathbf{g}\,,\ t \in \{1, 2, \ldots\} \tag{38}$$

Then, from equations (36),

$$\Delta\mathbf{w}(0) = -\eta\mathbf{g}. \tag{39}$$

From (37) and (38),

$$\Delta\mathbf{w}(1) \approx -\eta\mathbf{g} + \mu\Delta\mathbf{w}(0) \tag{40}$$

$$\Delta\mathbf{w}(1) \approx -\eta\mathbf{g}(1+\mu) \tag{41}$$

$$\Delta\mathbf{w}(2) \approx -\eta\mathbf{g} + \mu\Delta\mathbf{w}(1). \tag{42}$$

Substituting (40) into (41),

$$\Delta\mathbf{w}(2) \approx -\eta\mathbf{g} + \mu[-\eta\mathbf{g}(1+\mu)] \tag{43}$$

$$\Delta\mathbf{w}(2) \approx -\eta\mathbf{g}(1+\mu+\mu^2). \tag{44}$$

In general, for $t \geq 0$,

$$\Delta\mathbf{w}(t) \approx -\eta\mathbf{g}\left(\sum_{s=0}^{t}\mu^s\right) = -\eta\left(\frac{1-\mu^{t+1}}{1-\mu}\right)\mathbf{g} \tag{45}$$

Taking the limit of equation (45),

$$\lim_{t\to\infty}\Delta\mathbf{w}(t) \approx \frac{-\eta}{(1-\mu)}\mathbf{g}. \tag{46}$$

Thus, in shallow regions of weight space, where the gradient does not change much from one iteration to the next, the momentum term increases the effective learning rate by an approximate factor of,

$$\left(\frac{1}{1-\mu}\right) \tag{47}$$

over gradient descent alone. In steep regions of weight space, on the other hand, where gradient descent tends to oscillate, consecutive gradients will evaluate such that,

$$\nabla E[\mathbf{w}(t+1)] \approx -\nabla E[\mathbf{w}(t)] \tag{48}$$

so that the momentum term from one step to the next will tend to cancel each other out (see, for example, Figure 7.6 on pp. 268 in [1]). In other words, in steep regions of weight space, the effective learning rate is approximately the same with and without momentum.

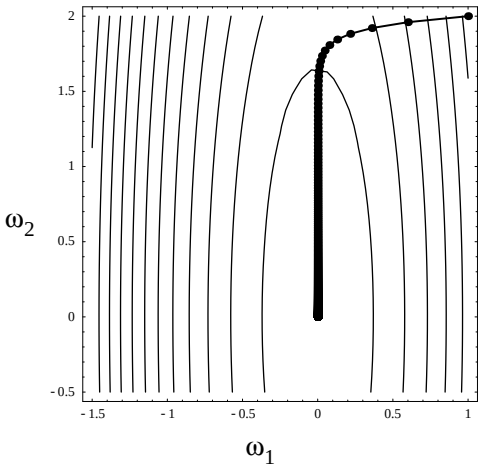
Thus, the main advantage of including a momentum term in neural network training is that it can increase the rate of convergence in shallow regions of weight space. The main drawback is that it requires that another parameter (in addition to the learning rate) be hand-picked for a particular neural network training problem. If μ is not carefully chosen, momentum can do more harm than good, destabilizing the convergence of basic gradient descent. Figure 2, for example, shows the trajectory in weight space for different combinations of (η, μ) for the simple quadratic error surface in equation (3) and equivalent initial weights and terminating condition as in Figure 1. Note from Figure 2 that while momentum can improve the rate of convergence, it can also destabilize the trajectory in weight space if μ is set to be too high.

C. Conclusion

In the neural network community, the addition of momentum to gradient descent has been, by far, the most popular modification to gradient descent. When carefully chosen, momentum can help improve convergence. At the same time, it introduces another parameter that has to be fine tuned for particular training problems.

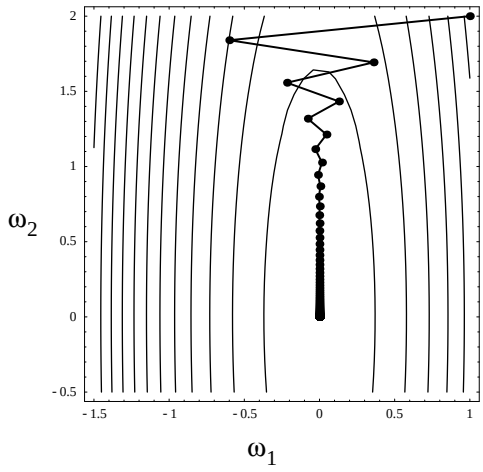
There have also been many other heuristic attempts to improve gradient descent. A selection of these is surveyed in [1] (section 7.5.3, pp. 268-72) and [2]. Most of these methods are not, however, well justified theoretically and often treat weights as independent from one another. This is often a very bad assumption that is

$\eta = 0.01, \mu = 0.0$



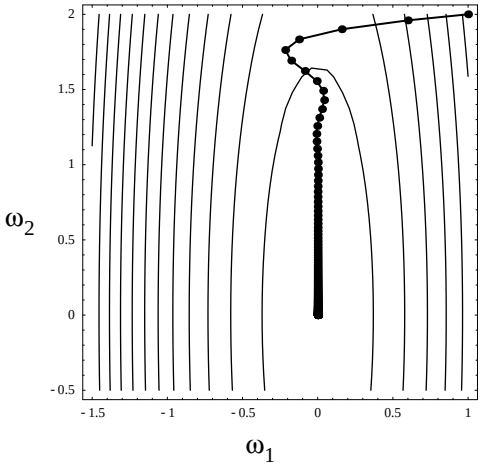
719 steps

$\eta = 0.04, \mu = 0.0$



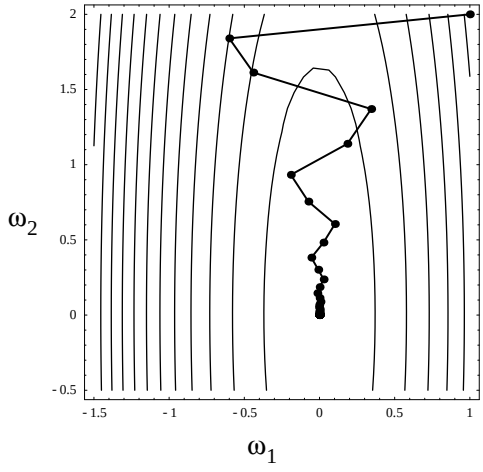
175 steps

$\eta = 0.01, \mu = 0.5$



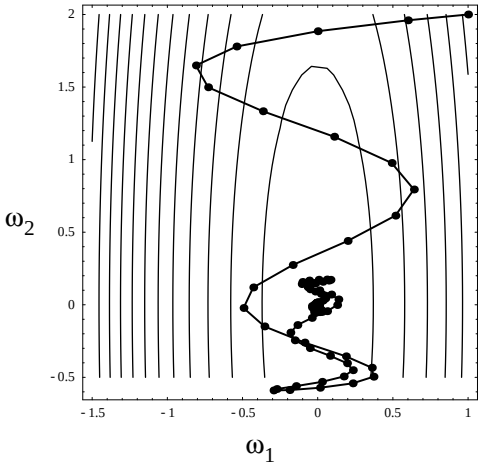
341 steps

$\eta = 0.04, \mu = 0.5$



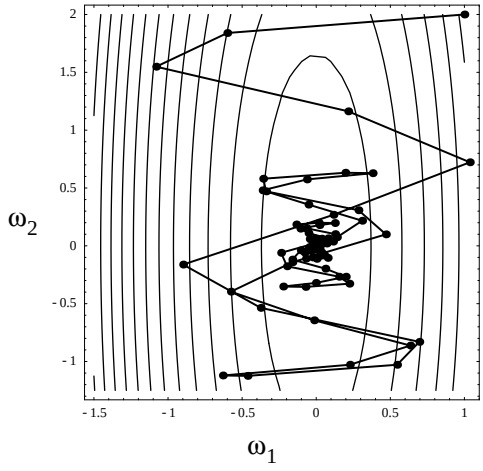
60 steps

$\eta = 0.01, \mu = 0.9$



266 steps

$\eta = 0.04, \mu = 0.9$



272 steps

Figure 2

the root of many neural network training problems. Therefore, we now begin to examine the neural network training problem in a more principled fashion. This examination will ultimately lead us to the *conjugate gradient algorithm* for training neural networks.

3. Steepest descent

A. Introduction

Recall that in the gradient descent algorithm for training neural networks, we update weights at step t according to,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t) \tag{49}$$

where,

$$\Delta\mathbf{w}(t) = -\eta\nabla E[\mathbf{w}(t)], \tag{50}$$

$\mathbf{w}(t)$ are the weights of the neural network at step t, η is a fixed learning rate, and $\nabla E[\mathbf{w}(t)]$ is the gradient of the neural network error E with respect to the training data evaluated at $\mathbf{w}(t)$. One obvious modification to basic gradient descent (which we will refer to as *steepest descent*) is to proceed as follows. First, set a *search direction* $\mathbf{d}(t)$ in weight space,

$$\mathbf{d}(t) = -\nabla E[\mathbf{w}(t)] \tag{51}$$

as we do in gradient descent. Second, minimize,

$$E(\eta) \equiv E[\mathbf{w}(t) + \eta\mathbf{d}(t)] \tag{52}$$

with respect to η, such that,

$$E(\eta^*) \leq E(\eta), \ \forall\eta. \tag{53}$$

Finally, set the new weights to be,

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta^*\mathbf{d}(t) \tag{54}$$

That is, rather than have η be fixed for all training iterations, we automatically set η at each step to give us the minimum along the direction $\mathbf{d}(t) = -\nabla E[\mathbf{w}(t)]$. This one-dimensional minimization can be done without computation of the derivative $\partial E/\partial\eta$ through a procedure known as *line search* or *line minimization*. As we show below, a line search requires only that we be able to evaluate $E(\eta)$ for different values of η. Since $\mathbf{w}(t)$ and $\mathbf{d}(t)$ in equation (52) are known vectors, this is not a problem.

The steepest descent algorithm depends on an accurate one-dimensional line search or minimization during each training step. A successful line search is usually performed in two steps. First, we need to bracket the minimum. Second, we need to use that bracket to zero in on that minimum. Below, we discuss each of these steps in somewhat greater detail.

B. Bracketing the minimum

In steepest descent, we want to find the value η^* of the learning parameter that minimizes $E(\eta)$,

$$E(\eta) = E[\mathbf{w}(t) + \eta\mathbf{d}(t)] \tag{55}$$

$$E(\eta^*) \leq E(\eta), \ \forall\eta, \tag{56}$$

where $\mathbf{w}(t)$ is the current set of weights in the neural network, and $\mathbf{d}(t) = -\nabla E[\mathbf{w}(t)]$. In order to perform this minimization without computing $\partial E/\partial\eta$, we first need to find three values of η, $\{a, b, c\}$, such that,

$$E(a) > E(b) \text{ and } E(c) > E(b). \tag{57}$$

In other words, we need to *bracket the minimum*. Figure 3, for example, shows three points that satisfy (57) and (57) for a sample function $E(\eta)$.

Below, we describe a simple algorithm for bracketing the minimum of an error functions $E(\eta)$ of form (55).

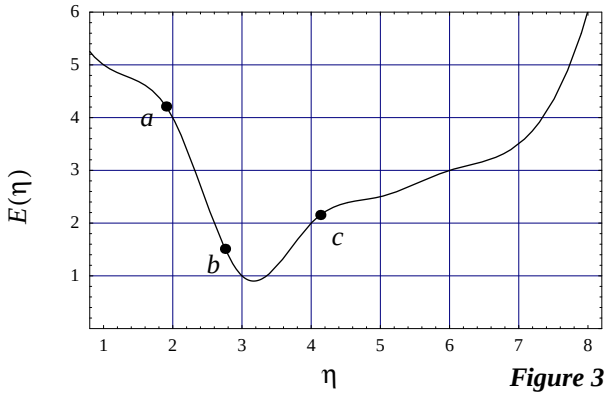


Figure 3

1. Let $a = 0$. Let $b = \varepsilon$, where ε is some small positive constant. Note that since we are performing the steepest-descent line search along the negative gradient, $-\nabla E[\mathbf{w}(t)]$, our initial choice of b guarantees that $E(a) > E(b)$.

2. Let $c = k(b-a) + a$, where $k > 1$.

3. If $E(c) > E(b)$, then we are done; else, let $a = b$ and $b = c$. Repeat step 2.

Note that step 1 requires one new function evaluation (i.e. $E(\varepsilon)$), since $E(0)$ has already been computed during the forward pass of the backpropagation algorithm. Each iteration of steps 2 and 3 then requires only one additional function evaluation of $E(\eta)$.

One popular choice for k in step 2 is the *golden ratio*; that is,

$$k = \frac{1+\sqrt{5}}{2} \approx 1.61803 \tag{58}$$

which tends to generate "nice" bracketing intervals (a, b) and (b, c). More on this a little later. Figure 4, for example, illustrates the above bracketing procedure for a sample one-dimensional function.[1]

Let us look at one more example. Consider the two-dimensional error function,

$$E(\omega_1, \omega_2) = 1 - \exp(-5\omega_1^2 - \omega_2^2) \text{ (plotted in Figure 5)} \tag{59}$$

and initial weights $(\omega_1, \omega_2) = (1, 2)$. For this function, the gradient is given by,

$$\nabla E(\omega_1, \omega_2) = [10\omega_1 \exp(-5\omega_1^2 - \omega_2^2), 2\omega_2]\exp(-5\omega_1^2 - \omega_2^2) \tag{60}$$

so that,

$$E(\eta) = 1 - \exp(-5(\omega_1 - 10\omega_1\eta\exp(-5\omega_1^2 - \omega_2^2))^2 - (\omega_2 - 2\omega_2\eta\exp(-5\omega_1^2 - \omega_2^2))^2). \tag{61}$$

At $(\omega_1, \omega_2) = (1, 2)$, equation (61) evaluates to,

$$E(\eta) = 1 - \exp(-5(1 - 10\eta\exp(-9))^2 - (2 - 4\eta\exp(-9))^2). \tag{62}$$

Figure 6 plots equation (62) and the three bracketing values $\{a, b, c\}$ that were found using the above algorithm.

C. Line minimization

Given the three bracketing values $\{a, b, c\}$ of η, we can now zero in on the minimum using the following simple algorithm:

---

1. *In Figure 4, we use the additional mechanism of parabolic interpolation to generate the first value of c. For details, consult section 10.1 in [3].*

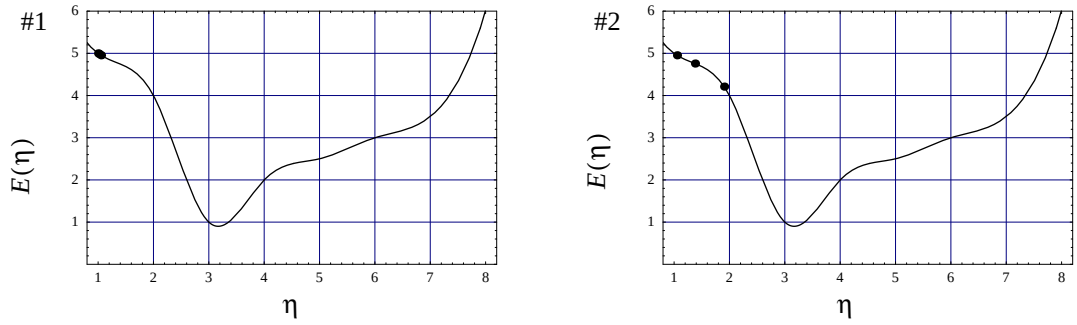
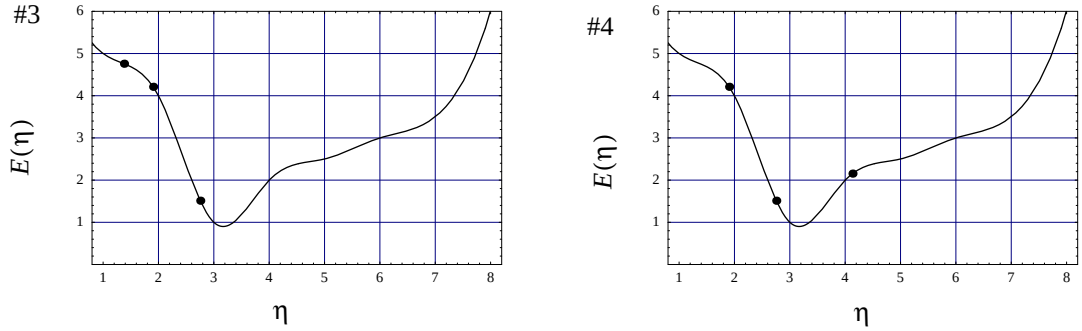


Figure 4



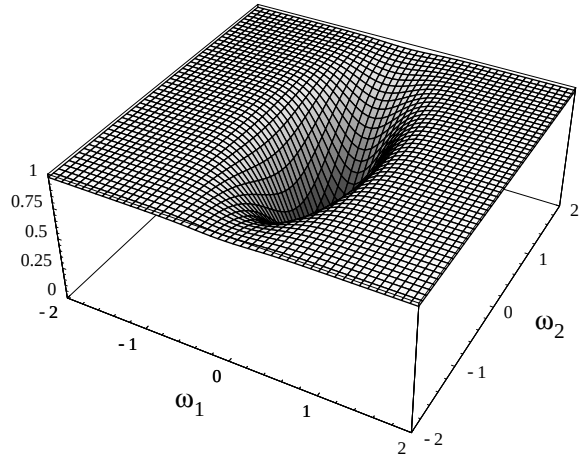


Figure 5

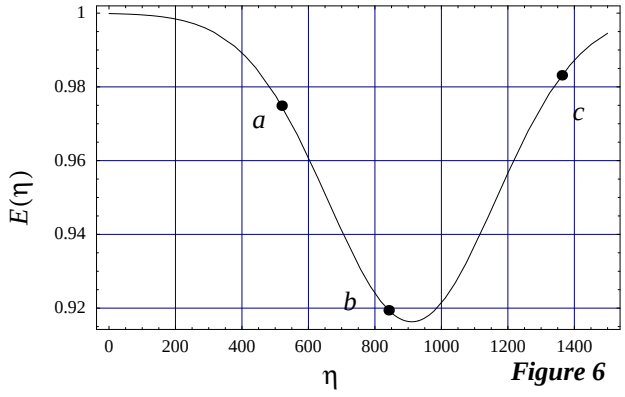


Figure 6

1. Pick a value of $\eta = x$ in the larger of the two intervals (a, b) and (b, c).

2. If (a, b) is the larger interval, then set the new bracketing values to:

$$\{x, b, c\} \text{ if } E(x) > E(b), \text{ (that is, set } a = x\text{), or} \tag{63}$$

$$\{a, x, b\}, \text{ if } E(b) > E(x), \text{ (that is, set } c = b \text{ and } b = x\text{).} \tag{64}$$

Else, if (b, c) is the larger interval, then set the new bracketing values to,

$$\{a, b, x\} \text{ if } E(x) > E(b), \text{ (that is, set } c = x\text{), or} \tag{65}$$

$$\{b, x, c\}, \text{ if } E(b) > E(x), \text{ (that is, set } a = b \text{ and } b = x\text{).} \tag{66}$$

3. Iterate steps 1 and 2 until $(c - a) < \theta$, where θ is some small number (e.g. 10^{-6}).

4. Set,

$$\eta^* = \frac{(c + a)}{2}. \tag{67}$$

In the above procedure, we would like to pick x so that, on average, we reduce the bracketing intervals by the largest amount possible for each iteration. Without loss of generality, let us assume that (b, c) is the larger of the two intervals. We then need to consider how much we reduce the bracketing interval for both possibilities,

$$E(x) > E(b), \text{ and,} \tag{68}$$

$$E(x) < E(b). \tag{69}$$

Let,

$$w = \frac{(b - a)}{(c - a)} \text{ and} \tag{70}$$

$$1 - w = \frac{(c - b)}{(c - a)}. \tag{71}$$

Also let,

$$z = \frac{(x - b)}{(c - a)} \tag{72}$$

Depending on whether condition (68) or (69) is true, our new interval lengths will be either,

$$(w + z)(c - a) \text{ (if } E(x) > E(b)\text{), or,} \tag{73}$$

$$(1 - w)(c - a) \text{ (if } E(x) < E(b)\text{).} \tag{74}$$

In order to minimize the worst case possibility, we should set equations (73) and (74) to be equal so that,

$$(w + z)(c - a) = (1 - w)(c - a) \tag{75}$$

$$z = 1 - 2w \tag{76}$$

[Note that since we assumed that (b, c) is the larger interval, $w < 1/2$ and $z > 0$.]

Now, we also expect that the optimal ratios of smaller to larger interval lengths be the same from one iteration to the next; in other words,

$$\frac{(x - b)}{(c - b)} = \frac{(b - a)}{(c - a)} \tag{77}$$

$$\frac{z}{1 - w} = w. \tag{78}$$

Substituting equation (76) into equation (78),

$$\frac{1-2w}{1-w} = w \tag{79}$$

$$w^2 - 3w + 1 = 0 \tag{80}$$

$$w = \frac{1}{2}(3-\sqrt{5}) \approx 0.381966 \tag{81}$$

Thus, the optimal ratio of smaller to larger interval lengths is given by (81). This means that the value of x should be chosen so that,

$$x = 0.381966(c-b)+b \tag{82}$$

when (b, c) is the larger interval, and,

$$x = b - 0.381966(b-a) \tag{83}$$

when (a, b) is the larger interval. With the value for x in equations (82) and (83), we are guaranteed to reduce the interval length by a factor of,

$$\frac{1}{k} \approx 0.61803 \tag{84}$$

where k is the golden ratio defined in equation (58). Because of this, this line search algorithm is known as the *golden section search*. Note that each step of this algorithm requires only one function evaluation of $E(\eta)$ (namely, $E(x)$) and at no time do we have to compute the derivative $\partial E/\partial\eta$.

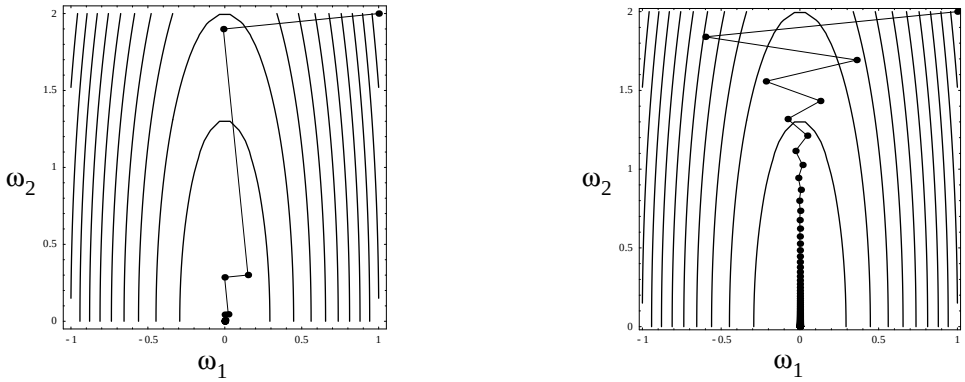
The rate of convergence for this algorithm can be further improved through parabolic interpolation (Brent's method). For details, see [3].

D. Quadratic error function example

Let us reconsider the simple, two-dimensional quadratic error surface that we looked at last time,

$$E = 20\omega_1^2 + \omega_2^2 . \tag{85}$$

For this error surface and from initial weights $(\omega_1, \omega_2) = (1, 2)$, Figure 7 below plots the trajectory in weight space of the steepest descent algorithm, and the gradient descent algorithm (with a near optimal learning rate of $\eta = 0.04$). Each trajectory is superimposed on top of a contour plot of E and is stopped when $\sqrt{E} < 10^{-6}$. Note that steepest descent converges significantly faster than gradient descent.



steepest descent
15 steps to convergence

Figure 7

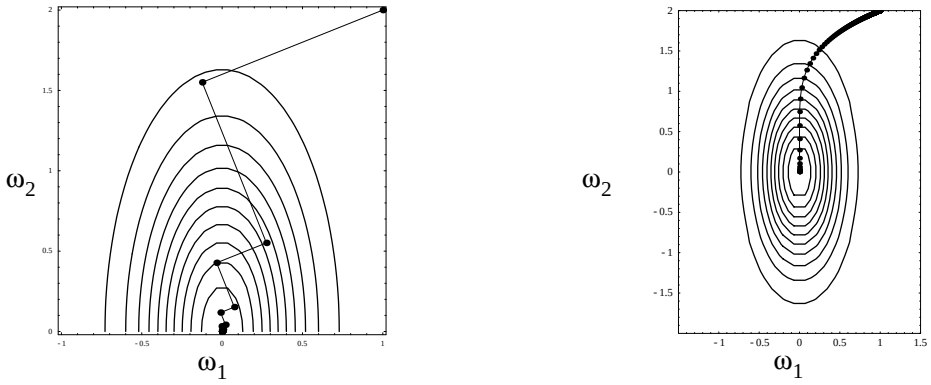
gradient descent $(\eta = 0.04)$
175 steps to convergence

E. Nonquadratic example

Here we again consider the simple nonquadratic error surface,

$$E = 1 - \exp(-5\omega_1^2 - \omega_2^2). \tag{86}$$

For this error surface and from initial weights $(\omega_1, \omega_2) = (1, 2)$, Figure 8 below plots the trajectory in weight space of the steepest descent algorithm, and the gradient descent algorithm (with a near optimal learning rate of $\eta = 0.2$). Each trajectory is superimposed on top of a contour plot of E and is stopped when $\sqrt{E} < 10^{-6}$. Once again, the steepest descent algorithm converges significantly faster than gradient descent.



steepest descent
24 steps to convergence

Figure 8

gradient descent $(\eta = 0.2)$
456 steps to convergence

F. Computational complexity

The above examples show that steepest descent generally converges in fewer steps than gradient descent with a fixed learning rate. Because of the line minimization procedure in each step, however, each steepest descent step requires more computations; that is, each line minimization requires that we evaluate the error function for multiple values of η. To compute the gradient in a neural network takes approximately $5NW$ computations, where N is the number of training patterns, and W is the total number of weights [1]. To compute the error for a specific set of weights takes approximately $2NW$ computations. Assuming approximately 10 additional error evaluations for each line minimization, one steepest descent step therefore takes approximately,

$$5NW + 10(2NW) = 25NW \text{ computations/step (steepest descent).} \tag{87}$$

Gradient descent, on the other hand, requires only that the gradient be computed, so that one step of the gradient descent algorithm requires approximately,

$$5NW \text{ computations/step (gradient descent).} \tag{88}$$

Thus the computational cost of one steepest descent step is approximately 5 times that of one gradient descent step. Even with that additional computational complexity, however, steepest descent is still more efficient for our examples above.

G. Orthogonality of consecutive steps

Note from both Figures 7 and 8 that consecutive search directions in the steepest descent algorithm are orthogonal. This is so because at each step of the steepest descent algorithm, we minimize the error along our current search direction $\mathbf{d}(t) = -\nabla E[\mathbf{w}(t)]$. In other words, the gradient in the direction of $\mathbf{d}(t)$ will have been made zero at our new weight values $\mathbf{w}(t+1)$. As an example, let us verify this property for the first step of the steepest descent algorithm in our quadratic error surface example above. For this error surface [equation (85)], the current weight vector is given by,

$$\mathbf{w}(t) = (\omega_1, \omega_2) \tag{89}$$

and the search direction at each step is given by,

$$\mathbf{d}(t) = -\nabla E[\mathbf{w}(t)] = -(40\omega_1, 2\omega_2) = (-40\omega_1, -2\omega_2) \tag{90}$$

Thus, from equation (52),

$$E(\eta) = E[(\omega_1, \omega_2) + \eta(-40\omega_1, -2\omega_2)] \tag{91}$$

$$E(\eta) = E[(\omega_1 - 40\eta\omega_1), (\omega_2 - 2\eta\omega_2)] \tag{92}$$

$$E(\eta) = 20(\omega_1 - 40\eta\omega_1)^2 + (\omega_2 - 2\eta\omega_2)^2 \tag{93}$$

With initial weights,

$$\mathbf{w}(1) = (1, 2), \tag{94}$$

equation (94) simplifies to,

$$E(\eta) = 20(1 - 40\eta)^2 + (2 - 4\eta)^2 \tag{95}$$

Instead of doing a line minimization, equation (95) is simple enough that we can solve for η^* explicitly:

$$\partial E / \partial \eta = 40(1 - 40\eta)(-40) + 2(2 - 4\eta)(-4) = 0 \tag{96}$$

$$\eta^* = \frac{101}{4002} \tag{97}$$

Thus, our new weight values are given by,

$$\mathbf{w}(2) = (1, 2) - \frac{101}{4002}(40, 2) = \left(-\frac{19}{2001}, \frac{3800}{2001}\right) \tag{98}$$

Let us now denote $\mathbf{g}(t) = \nabla E[\mathbf{w}(t)]$. From equation (89),

$$\mathbf{g}(1) = (40, 4), \tag{99}$$

$$\mathbf{g}(2) = (-760/2001, 7600/2001), \tag{100}$$

so that,

$$\mathbf{g}(1) \cdot \mathbf{g}(2) = 0. \tag{101}$$

Since the dot product of $\mathbf{g}(1)$ and $\mathbf{g}(2)$ is equal to zero, the two gradient vectors are orthogonal.

H. Conclusion

Note that having consecutive search directions be orthogonal is not a good thing. Progress made towards the minimum in one step is partially undone in the very next step. In other words, steepest descent constructs consecutive search directions that interfere with one another. As we shall see over the course of the next several lectures, there is a much better way to use gradient information to construct *non-interfering search directions* on a locally quadratic error surface. This will lead us to the *conjugate gradient algorithm*, which requires very little additional computation/step beyond that required in steepest descent, yet typically converges in many fewer steps.

[1] C. M. Bishop, “Chapter 7: Parameter Optimization Algorithms,” *Neural Networks for Pattern Recognition*, Oxford University Press, Oxford, 1995.

[2] S. E. Fahlman, *An Empirical Study of Learning Speed in Back-Propagation Networks*, CMU-CS-88-162, Technical Report, Carnegie Mellon University, School of Computer Science, 1988.

[3] W. H. Press, et. al., *Numerical Recipes in C: The Art of Scientific Computing, 2nd. ed.*, pp. 397-405, Cambridge University Press, Cambridge, 1992.